DESIGN AND IMPLEMENTATION OF THE LANGUAGE PILOT/360

BY

Raymond Douglas Rohn

June, 1977

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

Report No. UIUCDCS-R-77-879

DESIGN AND IMPLEMENTATION OF THE LANGUAGE PILOT/360

BY

RAYMOND DOUGLAS ROHN

A.B., Princeton University, 1972

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1977

Urbana, Illinois

Acknowledgement

There have been many people who have influenced my work during the several months involved in this project. I would like to express my sincere thanks to all of them. It is difficult to mention all by name, but I would like to single out some individuals for their special efforts.

I would like to thank my thesis adviser Dr. H. George Friedman, Jr. for providing the guidance for the work contained in this thesis. I am especially thankful for the freedom I was given to learn from my mistakes, of which I made many. But, whenever I needed to be pulled back on track, Dr. Friedman was there.

I would certainly be remiss if I did not thank Dr. Axel Schreiner for suggesting this project and providing much needed assistance in its initial phases. I learned much of my programming style by observing Axel in action. I hope I have done justice to his ideas in this implementation of PILOT.

Lastly, I wish to express my thanks to my wife, Sharon, and my daughter, April. Although they do not understand the material in this report they understood the many long hours that I was unable to spend with them because of it. With out their support these past two years none of this would have been possible, and it is to them I dedicate this thesis.

R. D. R.

Table of Contents

1 INTRODUCTION

A tried and proven method of instruction in the sciences that cuts across all disciplines is the laboratory (or hands-on) approach. This is also true for the software area of computer science as evidenced by the considerable use of computer facilities for instructional purposes by the computer science departments across the country. Although the practical aspect of software courses are seldom referred to as laboratories, there is hardly a course taught which does not require several programs to be implemented by the course's completion. Some software design and development courses, such as courses on compilers or operating systems, deal with such complex software systems that generally it is best if the students actually have an opportunity to implement a complete system as a class project. In such cases the instructor is faced with the problem of constructing a project which provides a sufficiently broad overview of the relevant concepts while being simple enough to implement within a semester.

Halstead [1] addressed this problem for courses on compiler and operating system implementation. His goal was to provide students with a compiler and an operating system that were rich enough to present realistic concepts and to support innovative student extensions while at the same time simple enough to understand and to actually implement. Halstead presented a small, systems-oriented language, called PILOT, for use in these laboratory projects and reported generally favorable student reaction.

Axel Schreiner extended PILOT for use in a course on operating systems given in the spring of 1976 at the University of Illinois. Since the course dealt only with operating systems and the PILOT compiler was not to be implemented by the students, several useful features (such as loops and sophisticated I/O) were added. The experience gained from that course suggested that PILOT could be used as an effective laboratory tool for an initial study of operating system concepts. However, PILOT still had some major deficiencies: no parameters in subroutine calls, all data was globally defined, no self-defining constants in the program body, etc. These shortcomings were probably due to the original intent of the PILOT language as an aid for a compiler and operating system course, and many of these extensions were left as student exercises once the basic compiler was understood. In a course only on operating

systems, however, these shortcomings detracted from a well structured implementation of an operating system.

As a result of this experience several more extensions were suggested and implemented by the author. The present version of PILOT is written in PL/I and is adapted for use on the IBM/360. The language specifications, as well as a discussion of the implications, can be found in [3, henceforth called the Manual]. This thesis will discuss the design and implementation of the major extensions to PILOT and will also discuss some uses of the language and some possible future modifications.

Before embarking on a description of the various extensions that have been added to PILOT, a brief list of the salient features of the language which make it useful as a teaching aid will be given. The following properties are each followed by a brief explanation.

1. Efficient. The features of PILOT have been chosen with ease of compilation and speed of object code in mind. All arithmetic is done in the general purpose registers and redundant fetches from storage are avoided.

2. One pass. The compiler is a one pass compiler and as such provides a ready example of how to handle forward references. The compiler provides a listing of generated code which the students may find useful

when learning about how external symbols and relocatable addresses are handled by loaders and linkage editors.

3. Reentrant and recursive. The program code generated by the compiler is reentrant, or pure code. The routines may also be called recursively. As such, the PILOT programs offer convenient examples of the advantages of separating data from code and the need for dynamic storage management.

4. Systems-oriented. The language has features to allow direct access to core memory for fetches, stores and transfers. In addition to decimal integers, octal and hexadecimal integers are also supported.

5. Embedded machine code. This feature is also part of PILOT's systems orientation. PILOT/360 will accept embedded IBM/360 machine code allowing for a closer contact with the machine than otherwise possible. This is a necessity in the efficient implementation of an operating system. This feature of PILOT offers a simple assembler capability which is faster than the more complex IBM assemblers. This also allows the instructor to only teach those aspects of the machine language that are deemed necessary.

6. Simple. The language has been kept simple for several reasons. It is intended that a student with previous programming experience can learn PILOT with minimal effort, thereby eliminating the need for the instructor to spend several weeks teaching the use of a more advanced systems-oriented language. Also, since it is anticipated that a team of students will implement a subset of PILOT on the project's system, the design of the PILOT/360 language and compiler have been kept simple for ease of understanding. Various debugging aids have also been incorporated into the compiler, such as a trace facility and a compiled comment option.

7. Versatile. Although the language has been reduced for simplicity, several control and I/O features have been provided to make PILOT very powerful. PILOT has such control structures as a looping mechanism, an _if then else_ statement, and a very useful decision table construct. PILOT also offers the full capabilities of IBM FORTRAN's IBCOM# I/O package. PILOT can be used to produce well structured code that is fairly self-documenting (once one adapts to the left-to-right nature of the language).

2 LANGUAGE EXTENSIONS

Under the direction of Halstead at Purdue University and Schreiner at the University of Illinois there have been many versions of PILOT implemented. It is the purpose of this chapter to describe some of the major features of this version of PILOT as presented in the Manual which distinguish it from the other versions, in particular, the version implemented by Axel Schreiner at the University of Illinois. The next five sections will each present an extension in the present version, and discuss some design and implementation issues relevant to that feature.

2.1 Global and Local Data

The previous version of PILOT required all data identifiers to be known globally (i.e., throughout the main routine and all subroutines). This detracted considerably from the structure of programs written in PILOT. It limited the amount of local program design that could be done within the context of each logical unit or routine.

As described in the Manual, PILOT/360 allows each routine to declare data identifiers local to itself. The scope of an identifier is determined by the compile-time environment. Data identifiers declared in the first (main) routine are known throughout all other routines, whereas identifiers declared in successive routines are known only within their respective defining routines. Local identifiers take precedence over identically named global identifiers. In addition to strengthening the link between data elements and their associated procedures, local data declarations also provide for the possibility of implicitly initializing local data upon each invocation of a subroutine rather than explicit initialization of various global variables. This possibility is realized by the run-time support package and is discussed further in chapter 4.

To understand the implementation of global and local data in the PILOT/360 compiler it is necessary to understand the register usage by the code it generates. The exact register usage is as follows:

14,15	subroutine linkage in OS/360
13	save area in OS/360
12 ... LOCAL	local data base registers
LOCAL-1 ... DATA	global data base registers
DATA-1 ... PROG	routine base registers
ACC ... 1	accumulator stack
0	scratch/parm info/return code

where LOCAL, DATA and PROG are determined by the run-time size of the local data area, global data area and procedure code for each routine (with ACC=PROG-1). The stack nature of PILOT/360 and the use of the accumulator stack is discussed in detail in the Manual. The use of register 0 will be covered more fully in chapter 4. One further point about the register usage that aided in implementation should be noted. The compiler can generate sixteen bit displacements for various identifiers and add the appropriate four bit base register representation to the high order bits of the displacement to produce a correct S-type address constant directly.

The contents of registers PROG through 13 are established upon entry to each routine. For each routine LOCAL, DATA and PROG can be different registers as explained above, therefore there must be a way of ensuring the integrity of the DATA base register since it contains a pointer to the global data. This is accomplished with the aid of the run-time support package which maintains the global data pointer in a standard location for entry linkage conventions. The run-time support package also allocates space for, and initializes, the local data at a fixed displacement from register 13. These linkage conventions are explained in more detail in chapter 4. An offspring of the method implemented to initialize local data was the addition of replication factors for initializing arrays. In this case

a desire for easier implementation lead to a better language design.

2.2 Parameters

One of the more restrictive aspects of the previous version of PILOT was the inability to pass parameters to subroutines. This severely reduced the structure of the resulting programs since global variables had to be used for all parameter passing. This limitation has been removed from the present PILOT language; up to eight parameters may be passed to a subroutine. As might be expected in a language that performs no type or bounds checking, in a subroutine call, actual parameters are not validated against formal parameter definitions for that particular subroutine. The actual calling protocol and its implementation will be covered in chapter 4.

Parameters may be scalars, array names or expressions. Scalars are passed by value but upon return they receive the last value the subroutine assigned them (by value result). Array names are passed by reference so that the subroutine works directly in the given array's storage area. Expressions (including constants and subscripted array names) are evaluated first and then passed as dummy scalars. The parameters are passed in fixed locations relative to register 13 of the calling routine. The definition of the

corresponding formal parameter actually determines how a value passed to the called routine is to be used (i.e., scalar or array) so care must be exercised when passing parameters. One result of there only being two data types in PILOT (array and scalar) is that the type of a parameter can be described in the formal parameter list in the routine label without declaring the parameter name in the declaration section of the routine.

2.3 Self-defining Constants

Another irksome aspect of the previous PILOT language was the inability to use literals (or self-defining constants) in the body of the program. Without this feature, all constants used in the program had to be declared as global constants in the data declaration section.

The PILOT/360 compiler now collects self-defining constants used in a routine and creates a literal pool at the end of the routine. The overall length of the code produced is lengthened only to the extent that the same constant is used in each of several routines, since global definitions are no longer required. Of course there are still cases where global constants would contribute to the clarity of a program. In order to implement this feature, some additional compile-time work space is needed to keep track of forward references to the literal pool. The symbol table still

requires an entry for each unique constant value. However, the removal of this inconvenience from the language is well worth the additional compiler storage required.

2.4 Character Strings

A common use for arrays in PILOT is for containing character strings. In the past, initialization of arrays with character information was cumbersome, requiring each array element to be initialized with the numeric equivalent of a character pair. Character strings were implemented in PILOT/360 primarily to make such initializations of arrays more convenient; thus character strings longer than two bytes can only appear in array initializations. Two-byte character strings, however, may be used anywhere a scalar may appear.

2.5 Decision Tables

To this point the extensions described were incorporated into PILOT/360 in order to remove some of its more glaring deficiencies. This last section will present an extension which was added primarily as an experiment. Schreiner has proposed a system implementation language, CLEOPATRA [4,5], which has as its major control structure a free format, extended entry decision table. Fixed format decision tables

have been in use in the business community for years now; see [2] and [6] for an overview. Schreiner [4] argues for the inclusion of decision tables into structured languages as a generalization of Hoare's case statement and suggests that they "... should be an ideal control structure, since they provide a high level of top-down execution discipline". A nearly equivalent version of the CLEOPATRA decision table has been implemented in the present PILOT/360 language.

In addition to the existing if then else (expression: statements ' statements;) structure, PILOT/360 now provides a decision table as its main structure for selective execution. A table consists of a decision part followed by an action part. The decision part consists of one or more decisions (expressions), each of which can set a single switch variable to be either true or false based on the logical evaluation of the decision expression. If there is a list of switches following a given decision expression, then the expression is treated as a pointer to the switch to be set true (a case statement); all other switches are set false. Following the decision part of the table is the action part; once it is entered, switches can not be altered. The action part consists of a series of actions (groups of statements) each preceded by a logical expression involving switches (a switch expression) which determines whether the following action is executed. Finally, a table may have an optional else part: an action to be executed if no other action is selected. It

should be noted that switches are the only identifiers whose scope of definition is less than their containing routine. Switch names were made unique to their defining table so that mnemonically meaningful names could be assigned in each table without any global implications.

At this point, a question that might arise is, "Why such a powerful control structure for such a simple-minded language?". There are two parts to the answer to this question. First, as stated at the beginning of this section, the decision table was implemented as an experiment to test its usefulness. PILOT is intended to be used in an introductory operating systems course by moderately sophisticated student programmers. As such, it will serve as a good setting to learn about the advantages and disadvantages of coding with decision tables in a systems-oriented language, however simple that language is. In that PILOT is already implemented, it has a distinct advantage over CLEOPATRA, and additionally, it will probably provide a larger 'test audience' for the decision table.

The second, and perhaps more defensive, answer to the question addresses itself to the implication that well structured programs are alien to simple languages. Admittedly, PILOT provides little in the way of maintaining the integrity of various control structures as would be typical of a more block structured language, allowing jumps to any statement within a control structure from any

statement outside of the structure. However, the fact that PILOT allows the 'harmful goto' does not constitute an endorsement of its uncontrolled use. As Schreiner [4] writes, "structured programming ... is a question of discipline in using one's intellect much more than a question of external discipline imposed by the designers and implementors of a language which may or may not provide a goto." PILOT is intended to be a simple, easily understood language (as is its compiler), but one will find that, with the addition of the decision table, PILOT is sufficiently rich in control structures to support a well disciplined programmer. The decision table will require some adjustment in ones coding style but in the hands of a good programmer it offers controlled top-down execution flow.

In the actual implementation of decision tables in the PILOT/360 language, each routine has an area (64 bytes long) at a fixed displacement from register 13 where its 'decision table' resides at run-time. During compilation each switch identifier encountered is assigned a fixed position (one byte) in the run-time table. Also, one position is set aside for each decision table to act as a monitor for the else action. Thus a sequence of nested decision tables can only declare a number of switches not more than 64 minus the depth of nesting. Disjoint decision tables (not nested in the source code) obviously can share the same positions in the run-time table for their switch values. One implication of

the present method of implementation is that every routine has a 64-byte area set aside for a table. Although this seems to be an optimistic view on the future use of decision tables, it actually is a result of the one pass nature of the compiler. To avoid having to resolve forward references to every switch name, it was decided to just fix the run-time table's location in a standard position. This approach was also taken with parameter values. Chapter 4 goes into more detail concerning the actual memory layout for these areas.

3 COMPILER STRUCTURE

Although PILOT/360 is a relatively simple language, its compiler is moderately complex with over 5500 lines of source code. This represents a significant increase over Halstead's original compiler of 250 source statements [1]. This increase can be attributed largely to the fact that the version of PILOT presented in the Manual is no longer intended as a teaching aid for compiler writing, so that many of the reasons for keeping the original compiler at a 'bare bones' level are no longer applicable. Some reasonably good error detection facilities have been added to the original version as have some very excellent debugging and instructional aids in the form of cross reference tables and a display of generated code. These features, along with several extensions that have been added to the basic language, have greatly increased the size and complexity of the present PILOT/360 compiler. The intent of this chapter is to provide an overview of the compiler's structure and some of its major components.

3.1 Design of the Compiler

The PILOT/360 compiler consists of the main driver routine and 45 external subroutines. A carryover from Halstead's compiler is that many of the compiler's variables are global and this has lead to some design difficulties (discussed in chapter 5) as the compiler has expanded. A complete description of the interconnections between all 45 subroutines would serve little else than to confuse the reader; however, the routines can be grouped into three functionally related areas: source scanning (lexical) routines, program structure (syntax and semantic) routines and service routines. The major control interconnections between these groups are shown in figure 1.

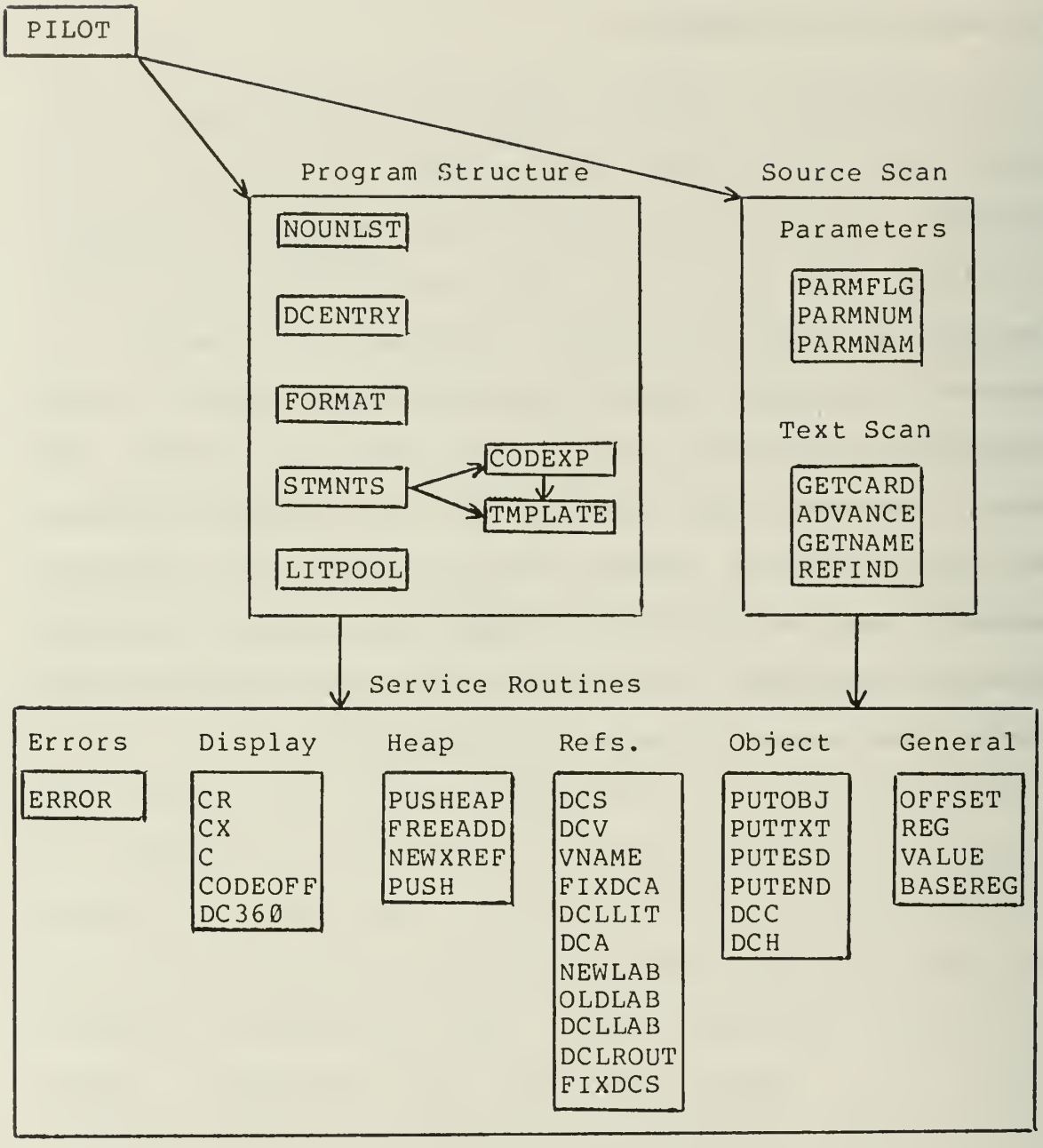


figure 1
Compiler Structure

The routines making up the program structure group constitute a major portion of the compiler and will be dealt with at greater length in succeeding sections. The main routine, PILOT, is primarily responsible for initialization

of internal tables and for sequentially driving the five routines in the upper left of figure 1 for each source routine to be compiled. Before discussing these routines and some of the compiler's main data structures in more detail, a brief overview of the other two functional groups will be presented.

The source scan group consists of two subgroups: the parameters subgroup responsible for receiving compile-time options from the source stream, and the text scan subgroup responsible for scanning the source text and producing tokens for the parsing routines of the program structure group. The parameters subgroup is called at compiler initialization and whenever a compiler command is encountered in the source text. In the text scan subgroup, GETCARD is responsible for the actual scanning; it reduces the text to a string of tokens using the following internal compiler code:

token	code	token	code	token	code
<	1	>	2	¬	3
/	4	=	5	,	6
(	7	:	8	)	9
-	10	;	11	+	12
*	13	.	14	$	15
'	16	?	17	!	18
@	19	%	20	#	21
&	22	|	23	¬=	24

=>	25	>=	26	¬<	26
<=	27	¬>	27	//	28
string	29	name	30	number	31

When names are encountered in the text, GETNAME finds them in the symbol table or locates an empty entry for a new name. The compiler is based on a straight-forward current operator -- operand -- next operator parsing scheme. ADVANCE provides the necessary capability to move along the stream of internal code produced by GETCARD and, as such, provides the primary interface between this group and the parsing routines.

Athough the service routines form a large part of the compiler, each individual routine is of minor interest so only a brief description of the six functional subgroups will be given. The subgroup which handles errors consists solely of the routine ERROR, by far the largest service subroutine. It is called by virtually every other routine and is responsible for formatting error messages and providing a local fix (usually skipping some text). The display routines are responsible for providing formatted output for such things as hexadecimal integers and for the IBM/360 code. The heap routines are responsible for managing the compiler's work space or heap. The use of the heap is described in section 3.2.1. The references subgroup provides the services necessary to maintain lists for forward references and to produce external (V-cons) and relocatable (A-cons) addresses.

The object routines format the generated code for output and write it to the object deck file. Finally, a group of miscellaneous routines provide general services such as establishing base registers or returning requested information from the source text.

3.2 Internal Data Structures

Much of the internal structure of a compiler is reflected in the design and structure of its major internal data structures. In the case of PILOT/360 there are two such structures: the symbol table and the heap. An understanding of these two structures will be invaluable to the understanding of the PILOT compiler.

3.2.1 The Symbol Table

The symbol table is the largest single structure in the compiler. It is allocated space at the beginning of each compilation according to a compile-time option (see the discussion of options in the Manual). The symbol table must be large enough to contain entries for every unique identifier (including literals) in the source program. The present compiler employs scatter storage addressing with linear rehashing, so failure to allocate more than enough area for the symbol table might result in degradation of

compilation times. The various parsing routines insert values in the symbol table. The various fields and their functions are given in the following table:

Field	Function
Name	First 6 characters of symbol
Block	Scope of identifier 1 .. global >1 .. local
Type	Identifier type -2 .. undefined routine -1 .. undefined label 0 .. data 1 .. label 2 .. routine 3 .. format 4 .. constant 9 .. switch 10 .. parm 11 .. literal
Address	Depends on type
Aux	Depends on type

The first three fields (name, block and type above) are fairly self-explanatory. Note that the block field is used

to determine the scope of an identifier. A new block is entered for every compiled routine or decision table. All identifiers (except formats and labels) declared in the first routine are known globally (i.e., block=1). Routine names are also assigned a block value of one. All other identifiers (including switches) are known only within their defining block (block>1). Once a symbol is added to the symbol table it is never removed (so that a cross reference table can be produced), so the block field also serves to distinguish identically named identifiers of differing scopes.

The following table gives an expanded view of the functions of the address and aux fields according to the type of symbol:

Type	Field	Function
undefined routine	Address	undefined routine chain
	Aux	absolute code chain in heap
undefined label	Address	undefined label chain
	Aux	relative code chain in heap
data	Address	relative to relevant data base
	Aux	-1 .. scalar otherwise .. length of array
label	Address	relative to program base
	Aux	unused

routine	Address	>Ø .. absolute program code
		<Ø .. external symbol
	Aux	unused
format	Address	relative to program base
	Aux	unused
constant	Address	relative to relevant data base
	Aux	value of constant
switch	Address	relative to register 13
	Aux	unused
parm	Address	relative to register 13
	Aux	-1 .. scalar
		otherwise .. length of array
literal	Address	literal chain
	Aux	reference chain in heap

A few words of explanation are in order for some of the above functions. The address field of undefined labels, undefined routines and literals are linked to three separate lists. These lists are used to generate messages and to create the literal pool at the end of each routine. The associated aux field for each of these symbols points to a chain maintained in the heap in order to resolve forward references. Most program addresses are relative to the current program base register but routine addresses are absolute locations within

the program object module. Note that for data and constant entries the address field is relative to the 'relevant' base register. This register is determined by the block field (i.e., whether the symbol is global or local).

3.2.2 The Heap

The heap consists of all the working storage for managing the cross reference table, forward references and the parsing routines' decision stacks. Space for the heap is allocated during the initialization phase and may be specified as a compile-time option. Considerable space is saved if a cross reference table is not requested. The heap consists of a linked list of standard records which may be allocated for various purposes by an appropriate service routine (see figure 1, section 3.1). A standard heap record has the following fields:

Next	link field
Aux	15 bits of information
Add	31 bits of information

where the aux and add fields are used differently according to the particular function of the record.

There are four major uses of the heap and they will be presented in order of decreasing demands placed on the storage available. First, if a cross reference listing has

been requested, a heap record is allocated for every occurrance of an identifier name in the source text. These records are not returned to the heap until compilation is completed. Second, heap records are allocated to separate lists, each headed by an undefined label, an undefined routine or a literal symbol entry. These records contain the information necessary to resolve forward references to these symbols. The records in the literal and undefined label lists are returned to the heap at the end of each compiled routine. Third, the statement parser (STMNTS) allocates a record from the heap onto its decision stack every time a compound statement (such as a loop, decision table or an I/O statement) is encountered. The record allocated is used to keep track of what kind of statement the parser is in and to resolve forward references peculiar to that type of statement. These records are returned whenever the parser finishes a statement. Finally, the expression parser (CODEXP) allocates heap records whenever it encounters a left parenthesis and restores one whenever it encounters a right parenthesis. These records are used to keep track of which run-time accumulator the code is being generated for and the surrounding syntax of the parenthesis.

3.3 Flow of Control

In section 3.1 an overview of the static structure of the PILOT/360 compiler was presented. The routines grouped together under the heading of 'program structure' in figure 1 of that section were lightly touched on. Since those routines constitute the major function (syntactic and semantic analysis) of the compiler, a section will be devoted to each of the major functional areas in that group of routines. The following five sections will present the routines in the same order as the driving routine (PILOT) invokes them during the compilation of a given PILOT source routine.

3.3.1 Noun List

Once the main routine finds a routine label (beginning with a '%') and resolves any forward references for that routine name, control is passed to the routine NOUNLST. NOUNLST is responsible for parsing the formal parameter list (if present) and the list of data declarations (required for every routine). Parsing of the formal parameter list is relatively straight-forward: an unsubscripted name denotes a scalar parameter and a subscripted name is an array parameter. The noun list is parsed using a 4 by 5 by 7 operand -- current operator -- next operator parsing table. A direct table implementation of this scheme was feasible due

to the limited number of operand types and operators permitted by the grammar for the noun list.

The noun list for both a main routine and a subroutine are parsed according to the same grammar rules (except main routines can not have formal parameters and subroutines can not declare external symbols) but the code generated in each case is distinctly different. NOUNLST essentially contains a separate code generator for each case.

The PILOT/360 compiler produces two separate object modules for each PILOT source program. The first module is the run-time copy of the global data area. Since this is only allocated once, NOUNLST can compile the global noun list (i.e., the first routine's data) directly as a core image. Beginning with the main routine's formats, all remaining code (program and data) is generated into the second object module. The noun list for each successive routine represents local data declarations. Since the areas for local data are dynamically acquired upon invocation of a subroutine, these areas must be initialized upon each invocation. Instead of generating a core image of the local data area, NOUNLST generates initialization records prior to the routine's program code and provides a pointer to these records for the run-time support package via a standard linkage convention in the routine's entry code. NOUNLST also provides some other miscellaneous information for the run-time support routine in the first twelve bytes of a data area. The details of this

information and the format of the initialization records will be given in chapter 4.

3.3.2 Entry

After the noun list is compiled the compiler must provide code for the entry linkage conventions of the routine. This code is generated by the routine DCENTRY. This routine is just a code generator and does no parsing so it might more rightly be called a service routine. However, it performs a service vital to the structure of the program and here presents an opportunity to give the details of PILOT linkage conventions.

Upon entering DCENTRY the following information is available: the size of the routine (from a compiler parameter), the size of the local and global data areas, and the location of the compiled initialization records for the routine. With this information DCENTRY can determine the values of the registers: PROG (the program base register), LOCAL (the local data base register), DATA (the global data base register), and ACC (the first accumulator register). (Refer to section 2.1 for a discussion on register usage.) With this information, DCENTRY generates the following linkage code for each routine:

```
STM     14,12,12(13)
L       15,24(15)
BALR    14,15
DC      CL14'+++ NAME +++'
DC      V(PILOT#)
DC      V(NAME$)   or   A(NAME$)
DC      V(IBCOM#)
LR      PROG,14
L       DATA,72(13)
LA      LOCAL,224(13)
```

Upon entry, a routine stores the calling routine's registers in its save area and immediately calls the run-time support routine, PILOT#. The details of the calling service of the run-time support package will be covered in a separate section (section 4.1), but some information concerning it is useful for understanding the above entry protocol.

The service routine expects the address of data initialization records (NAME$) to be at a displacement of 18 from register 14. (This is an external reference for the main routine since this is the address of the separate global data csect.) Before returning control, the service routine does the following: stores the address of global data in a fixed location, stores the value 4096 in register 15, and restores register 14 to point to the beginning of the called routine's code. The service routine then returns control to

a displacement of 36 from register 14 (i.e., past the V-cons). The user's routine can then load PROG, DATA and LOCAL with the appropriate addresses. If there is no local data (e.g., this is the first routine or there were no local data declarations) then DATA and LOCAL are the same register and the last statement above is not included in the entry code. If there is more than one base register (program or data), then DCENTRY generates code to use register 15 (containing 4096) to initialize the other base registers.

3.3.3 Formats

The PILOT language provides the services of the IBM FORTRAN I/O handler, IBCOM#. Every routine must have a format declaration section (possibly empty). A format declaration is used to associate an identifier with the beginning address of a FORTRAN format statement. A format itself is not checked for validity by the PILOT compiler; this fact is indicated by including the format in the comment section of the source card. Formats are compiled into the program code in order to discourage dynamic alteration of a format if it were treated as data. The parsing of format declarations is performed by the routine, FORMAT, and is very easy. A format label is verified for validity (i.e., it must be a new name) and it is inserted into the symbol table along with the present program code address. Then the information

following the comment delimiters of that card and all succeeding cards to the next labelled format is compiled without verification into the program code at that point. At the end of compiling the format section one forward reference needs to be resolved to provide for execution flow around the compiled formats.

3.3.4 Statements

The parsing of the body of a PILOT routine is performed by the two routines: STMNTS and CODEXP (figure 1). STMNTS identifies statements in the source text and generates code for the more complex, compound statements: routine calls, input/output, and decision tables. STMNTS also contains the crutch code parser. When a computation or expression (see the Manual for the grammar) is encountered, CODEXP is called. It parses expressions, loops and decisions and uses the services of the code generator, TMPLATE.

An examination of the grammar of PILOT will reveal that it is very easy to parse. The current operand and the next operator almost always determine what code generation is needed. The only additional information needed for compound statements is the type of statement presently being parsed. As stated in section 3.2.2, this information is maintained in a decision stack. Therefore, both the statement parser and the expression parser use a modified operand -- next operator

table parsing technique. In the actual implementation, the parsing is first done on the basis of a next operator table and then it is parsed on the basis of operand type by a series of decisions. This method was chosen for two reasons. First, a current operator -- operand -- next operator table would be very large (27x14x27), very sparse and would contain many redundant cross sections. Second, and perhaps more instrumental, the existing PILOT compiler had separated the statement and expression parsers making a direct table implementation extremely difficult.

Separating the statement and expression parsers, essentially providing a bi-level parser, did have some distinct advantages, however. Only a subset of the 27 operators and 14 operand types are permissible in each parser allowing for a reduction in the theoretical table size. As mentioned above, the parsing only depends on the next operator (or current operator in the case of CODEXP) and the operand type. This permits a reduction of one dimension (for the other operator) from the table. The resulting tables, although reduced in size, were still reasonably sparse with only a few distinct operand types permissible for each operator. For this reason and to ease modification of the existing code, the parsing based on operand type is performed by a (usually) short sequence of binary decisions. This approach results in parsing speeds close to those attainable by a straight table implementation because of the few

decisions actually required for each entry. The approach still allows for easy future modifications since it is still conceptually implemented by a table which is extremely sparse.

3.3.5 Literal Pool

Once the body of a routine has been compiled by STMNTS, the driving routine generates the return protocol for the routine and then calls LITPOOL to generate the literal pool for that compiled routine. Just like the DCENTRY routine, LITPOOL performs no parsing and might be considered a service routine. LITPOOL uses the literal list to code a half-word constant for each unique literal value at the end of the code for the routine just completed. Literals are assigned entries in the symbol table according to their compiled value so that equivalent literals for the same value share the same entry. As a value is placed at the end of the routine, all forward references to it are corrected.

3.4 Compiler Output

In the introduction, it was mentioned that one of the advantages of the PILOT compiler as an instructional aid is that its output format is designed to expose the internal workings of the compiler as much as possible. A good portion

of the PILOT compiler's code is devoted to providing highly readable output. The error messages are very meaningful (although short) for a compiler of such simple design. In addition to error messages, there are compiler options that can be set so that the student may see everything that the compiler generates from the IBM/360 symbolic assembler code to the hexadecimal display of the actual object code. This may serve as a useful aid for teaching about external symbols, relocation, reentrant code, forward references or just IBM/360 machine language. Actually seeing the generated code proves to be invaluable in debugging many systems-oriented programs.

Since a first course on operating systems tends to be a student's first exposure to diagnosing program bugs with the 'help' of pages and pages of a hexadecimal dump, the PILOT compiler generates output to ease this task. The symbol table, printed on request, lists the address of each identifier relative to its appropriate run-time base register in hexadecimal format. Debugging of PILOT programs is eased when both the text display and cross reference options of the compiler are enabled. In cases of dire need, selected statements may be traced during execution (see section 4.3). The printed output from the PILOT compiler will not eliminate the need for learning to use program dumps, but rather, it will help the student in understanding and using a valuable debugging aid provided by OS/360.

4 RUN-TIME SUPPORT

Most compilers provide for many of their features by generating calls to externally compiled routines existing in some system library. These external routines constitute the run-time support package. The PILOT/360 compiler is also supported by a run-time support package embodied in one external routine, called PILOT#, with three separate entry points. These three points correspond to the three services offered: subroutine calling, return from subroutine, and the statement tracing facility. Each PILOT routine is compiled with a V-con for PILOT# so the services can be invoked by entering PILOT# on displacement 0, 4 or 8 respectively. The other run-time support made available is the I/O handling facility of the IBM FORTRAN IBCOM# package.

4.1 Subroutine Call

A primary function of PILOT# is to facilitate the linking of PILOT subroutines. It is also the facility that allows PILOT subroutines to be called recursively. In order to call a subroutine, the calling routine first builds a parameter list as described in the next section and then

executes a branch and link to the subroutine. As shown in section 3.3.2, the called routine first saves the calling routine's registers and then transfers control to the calling service of PILOT#. The first twelve bytes of the called routine's data area contain information used by the PILOT# routine. A pointer to this information is provided to PILOT# at a displacement of 18 from register 14. These twelve bytes contain the following information filled in by the noun list parser at compile-time:

Start Byte	Field	Function
Ø	Size	Ø .. no initialization copy old global address
		1 .. no initialization create new global address
		2 .. main routine
		>2 .. initialization copy old global address Size-2 = length of initialization area
		<Ø .. initialization create new global address |Size-2| = length of initialization area

2	Parms	one 2-bit field per parm ØØ .. no parameter Ø1 .. scalar parameter 1Ø .. array parameter
4	Min	minimum area for getmain
8	Max	maximum area for getmain (Max=Min except for main)

PILOT uses the min and max fields to determine how much storage to allocate to the called routine. Each routine must be allocated a minimum of 224 bytes (192 for the main routine) as shown in figure 2 (on the following page). The 224 bytes for each routine provide the routine with the standard IBM save area, an area to build its parameter list for subroutine calls, 64 bytes for a decision table and an area to hold parameters received from the calling routine. Since a main routine can not receive parameters, these additional 32 bytes are not provided. Section 4.1.2 will cover the use of the parm list, dummy values and parameter areas of figure 2.

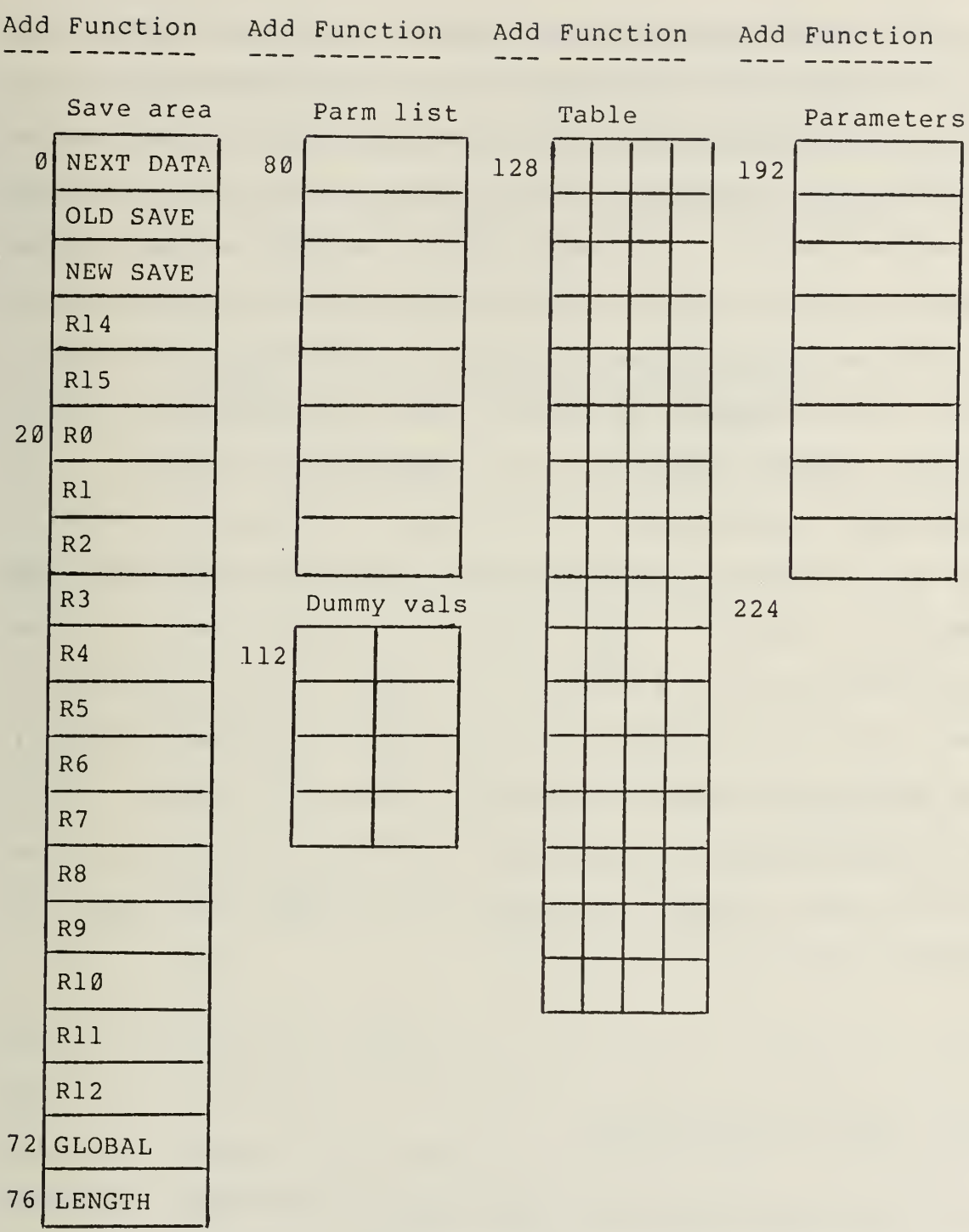


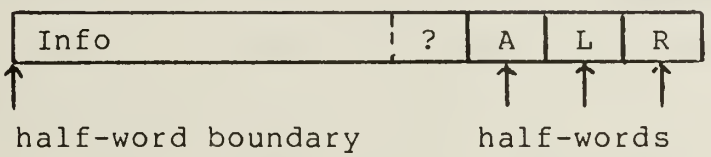
figure 2
Fixed Storage Requirements

For a PILOT main routine, PILOT# needs to allocate only the 192 bytes mentioned above since the associated data area for the main routine already exists just beyond the twelve bytes of size, parms, min and max information (thus requiring no initialization). At compile time, the user may specify a maximum initial area that PILOT# will attempt to acquire upon the invocation of the main routine. On future subroutine calls PILOT# attempts to allocate space from this area before issuing another getmain for additional storage. By specifying a sufficient initial area, considerable execution time may be saved. PILOT# uses the full-words NEXT DATA and LENGTH to maintain this run-time storage stack. If the called routine which PILOT# is servicing is a subroutine, then its local data is initialized starting at location 224, just after the fixed storage area of figure 2. Upon return to the called routine register 13 will point to its save area and its LOCAL and DATA registers may be loaded as shown in section 3.3.2.

4.1.1 Data Initialization

After acquiring necessary storage and updating register 13 for the called routine's save area, PILOT# must initialize the local data area. Of course, if the routine is a main routine or there is no local data, then only the address of the global area needs to be placed in a standard location for

the called routine to initialize its DATA register. If the size field described in section 4.1 indicates that initialization is required then the area following the size, parms, min and max fields contains a series of initialization records in the following format:



where the fields have the following meanings:

Field	Meaning
Info	Initial value
?	possible blank for padding
A	displacement in data area
L	length of Info
R	replication factor

These records enable PILOT# to initialize the local data area that it has acquired. Note that in the worst case of each local data element requiring a unique initial value, this technique will cost four times the storage needed to just provide a template of the local data area to be copied upon each invocation. Since several local data elements will probably not be assigned initial values and several contiguous elements (e.g., array elements) will probably be

assigned identical values, the above record format will generally save space over providing a complete copy of the data area. Finally, if this routine was an externally compiled PILOT subroutine, this newly created data area is the global data for its subroutines, so PILOT# updates the global data address before returning control to the requesting routine.

4.1.2 Parameter Passing

Another major function of the calling service provided by PILOT# is the copying of parameters from the calling routine to the called routine's parameter area (see figure 2 in section 4.1). Before calling a subroutine, the calling routine builds a parm list at location 80 relative to its register 13. Starting at this location the calling routine places the address for each parameter. Starting at location 112 there is space for eight dummy parameter values if needed. When the called routine receives control and requests the service of PILOT#, the parm list of the calling routine is used to copy the parameters into the parameter area of the called routine. If the associated formal parameter is a scalar, the actual value (using the locating address built by the calling routine) is copied; however, if the formal parameter is an array, the address of the array in the calling routine is copied directly. PILOT# uses the

parms field generated by NOUNLST to determine if a particular formal parameter is an array or a scalar.

4.2 Subroutine Return

The PILOT# support routine is also called when a PILOT subroutine wishes to return control to its calling routine. This is done primarily to release the storage that was acquired for this routine upon entry and to return the updated parameter values to the calling routine. The low order eight bits in the calling routine's register Ø (accessible in its save area) determine which parameter addresses in its parm list are to be used to return values to the calling routine. (The compiler generates code prior to a routine call which sets an appropriate bit in register Ø for each scalar data element passed as a parameter.) This is done because, on exit from a routine, PILOT# no longer 'knows' whether a parameter was a scalar or an array so the calling routine must mark each parameter (via register Ø) for which it expects to have a value returned. After returning the parameters, PILOT# restores the old registers (except register Ø which now contains the return code), restores the memory used by the routine to the run-time stack and then returns to the calling routine by a branch on register 14.

4.3 Trace Service

One minor service offered by the support routine is a trace facility. Although this service is available to the user programmer via the use of embedded assembler code, it is generally used only in conjunction with the 'F' (for flow) option for the compiler. As long as the 'F' option is enabled during compilation, the PILOT compiler will generate a call to the trace service of PILOT# prior to each statement it parses. During execution of the compiled PILOT program the trace service will use the facilities of IBCOM# to generate a formatted message identifying the current displacement in the code.

4.4 IBCOM#

In offering the services of the IBM FORTRAN IBCOM# I/O package, PILOT offers extremely powerful I/O capabilities. It is also an easily understood facility; every programmer has probably been exposed to the FORTRAN format statement at some point in his career. The author has found that the loop construct allowed inside PILOT/360 I/O statements gives the programmer the ability to produce highly formatted output with very compact notation.

The input/output statements in PILOT are meant primarily to provide for input/output services for unit record devices; however, IBCOM# provides services for all types of file organization (such as sequential, indexed, random access, etc.).

IBCOM# also provides many other services such as trace back facilities and direct return to the system on abnormal program termination. These types of services may prove extremely useful in systems applications. To this end, the address of the IBCOM# package is placed in a standard location in the entry code of each routine. To avail himself of these services, the programmer will have to set up the proper calling sequence using the embedded machine code facility. In summary, most programmers will feel very comfortable with I/O facilities offered by PILOT.

5 FURTHER CONSIDERATIONS

To this point, this thesis has presented the major extensions of the PILOT language and has described the compiler itself in some detail. The intent has been to provide a basis for the appreciation of the language and some issues related to its implementation. However, a discussion of a proposed laboratory tool would not be complete without some suggestions set forth as to possible uses for it. The first section of this chapter addresses itself to this issue. Also, any language suffers from deficiencies, and PILOT is no exception. Some of these shortcomings are intentional, due to time, cost and other considerations, and some are not. The second section of this chapter points out some of the known shortcomings and suggests possible solutions.

5.1 Suggested Uses

PILOT/360 is designed to be an easy-to-learn systems-oriented language. A student should be able to learn enough from an hour's reading of the Manual to program moderately well structured PILOT programs. The use of decision tables will probably require some amount of

additional training. The left-to-right nature of the language and the possibility of embedded assignments in expressions will take some familiarization, but can be used to produce some very clever, yet readable, code. As a practical tool in a course on operating systems, PILOT's embedded machine code facility is invaluable (assuming an IBM/360 machine is available). This feature allows for efficient use of the machine's capabilities needed in systems applications. The PILOT/360 language presently offers sufficient scope to support most course designs. Three different approaches are briefly outlined below.

One approach, which might offer the greatest possibilities for integration with an existing course, would be a general 'algorithmic' approach. That is, an instructor may choose to have the students implement various standard algorithms encountered in the implementation of operating systems (e.g., schedulers, interrupt handlers and I/O buffering). Such an approach could avoid IBM/360-specific solutions by not requiring the solutions to actually work on the host machine. This, of course, would eliminate the use of the embedded machine code feature, but, more importantly, it would blunt the intent of PILOT -- to make an actual machine more accessible for examination by the student. This approach, however, does offer a certain amount of realism for the student. Much of the systems programming done today is still done at the machine code level. Without becoming

engrossed in the particulars of a machine, students using PILOT can get a sense of programming with a relatively low-level language (with no data types or provisions for program security).

A second approach would involve using the present PILOT/360 compiler to implement a mini-operating system to run under the control of OS/360. Using the services of IBCOM#, input/output can be performed using any of the standard peripherals found with an IBM system. An area on disk could contain a library of PILOT 'user' programs and be managed by a 'filing system'. The PILOT language provides the features to allow a routine to act as an absolute loader into its own data area. Another routine (probably the main routine) would have to be provided to simulate interrupts from the 'job stream'. This approach would probably involve a considerable probing into the architecture of an IBM/360 machine and its operating system by the student without requiring the student to implement an entire system for any machine. The resulting system would, however, involve many of the concepts necessary for the understanding of other systems.

Another approach to using PILOT as an aid in an operating systems course would allow the students to implement an entire working system by carrying the above approach a step further. Instead of inserting an operating system on top of the existing OS/360, insert an 'entire

machine' into the IBM/360 and write an operating system for the new machine. This can be accomplished by writing a simulator (in PILOT of course) for the new machine. It would also be nice if the new machine had a PILOT compiler for it. With this in mind the designer of the new machine has an excellent opportunity to design a machine suited to the PILOT language. The result would probably be a stack machine with hardware support for segmentation (for program, local and global data areas). The designer can scale down the complexity of the machine while including features of instructional value (e.g., paging hardware and interrupt hierarchy).

This solution does require more preparation by the instructor and/or more laboratory time for the course. The simulator could be provided by the instructor or assigned to a student programming team. The instructor should provide at least a design for the new PILOT compiler (probably based on a subset of the PILOT/360 language). Another team of experienced student programmers could produce a small PILOT cross compiler (to run on the IBM/360 and produce code for the new machine) while the simulator was being coded. The remainder of the class would be involved in the design phase of the operating system. This would offer a class project that would be realisticly representative of a typical system design and implementation. As a preventative measure, the

instructor might provide the back up cross compiler for PILOT in case the student implementors run into problems.

Regardless of the approach chosen by the instructor, a laboratory course will take several semesters to evolve as the instructor adjusts the scope of the project. This is particularly true of the last approach considered. It may take several attempts to tune the design of the project and to accumulate a library of 'plug-in' modules to cover up rough areas in the project.

This section is not meant as a final word on instructional techniques or even on the use of PILOT; it has attempted to show a few of the possibilities for using PILOT as an instructional aid in a course on operating system implementation. The PILOT language is versatile enough to handle a wide spectrum of course structures.

5.2 Suggested Modifications

Although the present PILOT/360 compiler is complete, several possible modifications to the compiler or extensions to the language presented themselves in the course of implementation. This section presents a potpourri of suggestions in the form of a list. Before giving the list, a word to the prospective implementor is in order. The PILOT compiler has undergone one major revision and the 'patient'

barely survived the 'operation'. Any significant modifications to the compiler should probably be implemented in a completely redesigned version to keep from propagating the original design errors that are now enmeshed in the code of the present compiler. This section concludes with an admittedly incomplete list of problems along with some suggested solutions:

1. The present compiler runs in about 250-280K bytes of core, depending on the size of internal tables. This size can be reduced by employing the overlay facilities of the IBM linkage editor. All major routines were compiled separately as external procedures with this in mind.

2. The present PILOT language originally was to allow for function calls (i.e., a routine call returning a value) but an ambiguity was discovered in the grammmar after implementation had proceeded too far. The production rules for PILOT would not discern whether the following form is a label definition or a function (without parameters) as the lone operand in an expression beginning a decision:

   NAME :

   A slight change in the production rules for labels or routine calls is necessary to allow for function calls. The present compiler could be modified to accomodate this change; the present return code

feature was originally intended to provide for the return of function values. The run-time support program, PILOT#, would also have to be modified to handle nested function calls if they are to be allowed.

3. A useful feature for a systems-oriented language would be a location operator (or even pointer types). A simple operator could be of the following form:

[...]

where the brackets can only contain an identifier or a subscripted array name, so that

[X] => Y

would assign the address of X to the scalar Y. Other operations for pointer information and perhaps an introduction of pointer variables would be more useful; however, this extension would require a careful examination of the existing grammar.

4. The present arrangement for a fixed area required for each routine for its decision table and parameter passing may seem to be an extravagant waste of memory if the routine does not use a table or pass parameters. This could probably be avoided by modifying the run-time routine, PILOT#, to check whether the called routine will require any of these storage areas. In order to do this, the entry

linkage sequence will have to be changed (in DCENTRY) as will the information provided by NOUNLST at the beginning of each data area.

5. Some increased efficiency in the compiler's generated code might be gained by utilizing various optimization techniques although most of the code is reasonably efficient. However, most of the code for switch expressions could be simplified by using the logical operations available in the IBM/360 instruction set instead of implementing switch logical operators the same way as scalar logical operators. This would involve some changes in the code generator, TMPLATE.

6. The original PILOT compiler [1] was intended to be a self compiler. This means that the the original compiler's variables were all global and none of its routines had parameters. This design constraint has been carried through in successive implementations. The present PILOT/360 compiler contains numerous global variables and great care must be execised when changing any one routine. Future versions of the compiler should be careful to avoid the pitfalls of unwise use of global variables.

7. Another serious flaw in the present compiler is that code generation is not restricted to just one set of routines. This greatly restricts the portability of the compiler. This modification is not feasible for the present compiler but should be considered as a valid issue for any future versions.

8. One last suggestion for any future implementors concerns the parsing technique. An actual current operator -- operand -- next operator parsing technique is a reasonable method to pursue. This would require a combining of the functions now performed by the routines STMNTS and CODEXP. A redesign of the internal compiler code and representation of operand types may also allow for considerable compressing of the parsing table's size.

6 CONCLUSIONS

The objective of the current research has been to extend the PILOT language so that it might be more effective as a teaching aid for courses on the implementation of operating systems. In the course of modifying the existing PILOT compiler, a second objective was to improve its structure in several areas. In general, the research has met the first objective with some degree of success. In regard to the second objective, however, some problem areas still persist as mentioned in section 5.2.

The extensions to PILOT which have been described in this thesis have removed much of the awkwardness present in the previous version which detracted from its effectiveness as a teaching aid. Most of the design effort in this project has been directed towards providing constructs capable of supporting a more structured programming style. The author has coded examples of typical problems in the new PILOT language and has found that the addition of parameters, local variables and decision tables has greatly improved the ability to program in a top-down fashion. Many of the unique features of the language, such as the left-to-right evaluation of expressions and the lack of any

English-sounding key words, will require some familiarization before one will feel comfortable with the language. However, the features which have been added to PILOT along with its already existing systems orientation make it quite capable of supporting a structured implementation of a major systems project.

Many of the hours spent in implementing the present version of the compiler were spent on improvements to its structure. An intensive effort was made to isolate its various logical components and to clearly delineate the flow of control. The most productive aspect of this effort resulted in making most of the compiler's routines external. This considerably eased the work necessary to make further modifications. However, the decision to modify the existing compiler rather than rewrite it appeared more unwise as the months of implementation dragged on. It became apparent that the problems with global variables and the parsing tables mentioned in section 5.2 were intimately linked with the design of the entire compiler, and a complete redesign would be required in order to remove them in the future. The present compiler is structured well enough to facilitate an understanding of how it works and to accomodate some minor changes but the effort spent in redesigning it before any major modification is undertaken will be well worth the time.

Although section 5.2 presented many suggested modifications to the PILOT language, the next step before any modifications should be considered would involve the development of a course structure in which to use PILOT. The present version is the result of experience gained from one such course and any future modifications should probably be based on similar experience. It is hoped that the present version has removed enough difficulties from the language as to not need any major extensions for some time.

In summary, the present PILOT language contains enough novel features to make it an interesting language in its own right. It is an easy to use, systems-oriented language well adapted for instructional use and merits further examination.

REFERENCES

[1] Halstead, Maurice H., A Laboratory Manual for Compiler and Operating System Implementation, American Elsevier Publishing Co., Inc., New York, 1974.

[2] London, Keith R., Decision Tables, Aurebach Publishers, Inc., Princeton, N.J., 1972.

[3] Rohn, R. Douglas, "PILOT/360 Reference Manual", Technical Report UIUCDCS-R-77-875, Department of Computer Science, University of Illinois, Urbana, Illinois, 1977.

[4] Schreiner, Axel T., "A Proposal for Another System Implementation Language", Ph.D. Thesis, Department of Computer Science, University of Illinois, Urbana, Illinois, 1974.

[5] Schreiner, Axel T., "Comprehensive Language for Elegant Operating System and Translator Design", Technical Report UIUCDCS-R-74-646, Department of Computer Science, University of Illinois, Urbana, Illinois, 1974.

[6] ------, SIGPLAN special issue on Decision Tables, September 1971.

BIBLIOGRAPHIC DATA SHEET	1. Report No. UIUCDCS-R-77-879	2.	3. Recipient's Accession No.
4. Title and Subtitle: DESIGN AND IMPLEMENTATION OF THE LANGUAGE PILOT/360			5. Report Date: JUNE 1977 6.
7. Author(s): RAYMOND DOUGLAS ROHN			8. Performing Organization Rept. No.
9. Performing Organization Name and Address: Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois 61801			10. Project/Task/Work Unit No. 11. Contract/Grant No.
12. Sponsoring Organization Name and Address: Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois 61801			13. Type of Report & Period Covered: Master's Thesis 14.

15. Supplementary Notes

16. Abstracts

PILOT is a simple systems implementation language designed for use as a teaching aid in courses on operating system implementation. This report describes some of the major features of PILOT and discusses their implementation. It also proposes some uses for the language as a teaching tool. The language has a generalized decision table as its major control structure and it allows the use of embedded IBM/360 machine code.

17. Key Words and Document Analysis. 17a. Descriptors

Compilation
Compilers
Programming languages
Parsing
Machine-oriented languages
Operating systems
Recursive routines

17b. Identifiers/Open-Ended Terms

PILOT
Systems implementation languages
Decision tables

17c. COSATI Field/Group

18. Availability Statement	19. Security Class (This Report) UNCLASSIFIED	21. No. of Pages 63
	20. Security Class (This Page) UNCLASSIFIED	22. Price